A scheduler to foster data locality for GPU and out-of-core task-based linear algebra applications

Maxime Gonthier[a], Loris Marchal[b], Samuel Thibault[c]

[a]*University of Chicago & Argonne National Laboratory, 5730 S Ellis Ave, Chicago, 60637, Illinois, United States of America*
[b]*CNRS & ETS Montreal, 1100 R. Notre Dame O, Montreal, H3C 1K3, Quebec, Canada*
[c]*University of Bordeaux, 200 Avenue de la Vieille Tour, Talence, 33405, Nouvelle-Aquitaine, France*

Abstract

Hardware accelerators like GPUs now provide a large part of the computational power used for scientific simulations. Despite their efficacy, GPUs possess limited memory and are connected to the main memory of the machine via a bandwidth limited bus. Scientific simulations often operate on very large data, that surpasses the GPU's memory capacity. Therefore, one has to turn to **out-of-core** computing: data is kept in a remote, slower memory (CPU memory), and moved back and forth from/to the device memory (GPU memory), a process also present for multicore CPUs with limited memory. In both cases, data movement quickly becomes a performance bottleneck. Task-based runtime schedulers have emerged as a convenient and efficient way to manage large applications on such heterogeneous platforms. **We propose a scheduler for task-based runtimes** that improves **data locality** for out-of-core linear algebra computations, to reduce data movement. We design a data-aware strategy for both task scheduling and data eviction from limited memories. We compare this scheduler to existing schedulers in runtime systems. Using StarPU, we show that our new scheduling strategy achieves comparable performance when memory is not a constraint, and significantly better performance when application input data exceeds memory, on both GPUs and CPU cores.

Keywords: Scheduling, Memory-aware scheduling, Eviction policy, Tasks sharing data, Runtime systems, Data locality, GPUs, CPU

Email addresses: mgonthier@uchicago.edu (Maxime Gonthier), loris.marchal@ens-lyon.fr (Loris Marchal), samuel.thibault@u-bordeaux.fr (Samuel Thibault)

1. Introduction

High-performance computing applications, such as simulations for aeronautics, material strength, or seismology, continue to require more and more intensive computing power on ever larger amounts of data. This increasing computing demand is being met by using increasing parallelism, with large multicore processors and hardware accelerators such as GPUs. However, recent technology trends show a widening gap between peak compute speed and communication bandwidth as well as decrease in memory per gigaflop [1, 2]. With users trying to solve larger systems, it becomes common to encounter a situation where all input data of the problem cannot fit into the memory of the computing units (either CPUs or GPUs).

To avoid running out of memory, *out-of-core* computing has emerged, allowing the input data to stay on slow but large storage (typically disk) and to be loaded into the memory of the processing units (typically CPUs) whenever needed for the computation [3]. All cores of a multicore CPU share a common (limited) memory, as depicted on Figure 1. The bandwidth between the disk and the CPU memory is only of a few hundred MB/s, which has a strong impact on performance if data movement are not carefully handled to overlap them with computation. Similarly, one should avoid loading several times the same data but favor data reuse, by improving temporal data locality, i.e., by performing all computations on the same data before its eviction from the memory.

Recently, the trend has been to leverage GPUs in addition to CPUs, to achieve unprecedented computation speed as well as energy efficiency. GPU can achieve multiple teraflops in performances but are limited by a shared bandwidth of a few thousands MB/s. In this case, *out-of-core* computing consists in keeping the whole input data in the (larger) memory of the CPU, and loading data in the memory of a GPU only when it is needed for computation (see Figure 2). In such a distributed memory context, data reuse is even more crucial as one has also to focus on spatial data locality, that is to aim at gathering all computations on the same data on the same GPU. Hence, the gap between communication speed and computation is a strong performance bottleneck when computing on large data.

In this paper, we consider that these two out-of-core situations (multicore CPU with limited shared memory and GPUs with limited distributed memory) can be treated similarly, and we focus on any system with two levels of memory: one fast and limited memory devoted to computations (in red in Figures 1 and 2), and one large but slower memory devoted to storing the whole dataset (in green in Figures 1 and 2).

To harness the power of complex heterogeneous computing platforms, it has

become very common to use task-based programming, i.e. to express the application computation as a Directed Acyclic Graph (DAG), and let a dynamic runtime system such as OmpSs [4], PaRSEC [5], or StarPU [6] manage the execution of the task graph over such distributed and heterogeneous platforms. The burden of allocating data in memory, choosing task processing order and mapping is thus offloaded from the application programmer to the runtime system, in the form of a task scheduling problem. As the runtime system handles both data and tasks, it has the opportunity to minimize data movement. The runtime scheduler must solve the complex task of deciding the mapping of the tasks to the processing units to improve spatial data locality, as well as the ordering of the tasks themselves, to privilege the temporal locality of data, thus favoring data reuse and saving duplicate data transfers. In addition, this scheduler has to account for task priorities (some tasks may be more crucial to compute early than others) and task affinities (in case of a heterogeneous platform, some tasks have a larger acceleration factor on GPUs than others). Besides, it is also essential to trigger data transfers ahead of task execution (data prefetches) so that they can be overlapped by the computation of the previous tasks. Last but not least, when the embedded memory of the processing unit is full, the runtime has to carefully decide which data should be evicted from it, to make room for further data.

In this paper, our aim is to solve this challenging problem, that is we propose a **task scheduler for runtime systems managing several computing units, within a single node, with shared or distributed limited memory**. We also aim at managing data movement (loads and evictions). More precisely, we want to determine (i) the assignment of the tasks to the processing units to reach a good load balance and spatial data locality and (ii) the order in which tasks must be processed on each processing unit to optimize temporal data locality as well as maximize overlap between computations and data movement. Our solution is *data-centric*, trying to re-use data as much as possible: a processing unit is assigned all the "free" tasks, for which it already owns all input data. Then, when no more free tasks can be performed on a processing unit, we look for the data that, if transferred to its memory, would allow to assign many tasks on this processing unit, for a minimal transfer time. Our scheduling strategy also looks for a good trade-off between data locality and priority in the task graph. Finally, we design a custom eviction policy that takes advantage of the (limited) vision of tasks to be processed on a processing unit to avoid evicting data that will be useful in a near future.

In this paper, we make the following contributions:

- We propose DARTS, a Data-Aware Reactive Task Scheduler for out-of-

core computing in task-based runtime systems with a custom eviction policy;

- We implement DARTS within the StarPU runtime;
- We compare the performance of DARTS and existing schedulers on two classical linear-algebra benchmarks (namely Cholesky and LU factorization) and on two out-of-core settings: (i) multiple GPUs with distributed memory and (ii) a multicore CPU with shared memory;
- Our experiments demonstrate that our scheduler is able to achieve good performance even when the memory is very limited, whereas all existing schedulers see their performance drop. Besides, DARTS performs comparably to the best existing schedulers when memory is not limited.

This paper extends a preliminary work on the DARTS scheduler [7]. Although both versions use the same intuition for scheduling tasks, the earlier version of DARTS was limited to independent tasks, and hence was only tested on restricted datasets. The redesigned version of the DARTS scheduler is able to handle general task graphs as required by runtime systems. It also uses an improved eviction policy, and special care has been devoted to reducing its computational complexity. While the previous version of DARTS was only tested with artificially limited memory for the GPUs and thus with much smaller datasets, the present paper showcases experiments on significantly larger datasets using the full memory of the devices, which corresponds to real-life scenarios. Besides, the present paper also compares DARTS to many other schedulers from the literature (LWS, DMDAS and AP from DPLASMA/PaRSEC) and tests all strategies on both CPUs and three different kind of GPUs. This large comparison on large task graphs across various type of GPUs and CPUs was made possible thanks to the new design of DARTS, its improved eviction policy and its reduced complexity.

The paper is organized as follows. Section 2 presents previous work in the area. Section 3 details the scheduling problem in runtime systems. Section 4 presents existing schedulers for runtime systems; our scheduler is introduced in Section 5. Section 7 relates the experiments conducted to evaluate the performance of the schedulers on both multiple GPUs and CPU cores.

2. Related work

To address the problem of fitting a working set in a given amount of memory [8], researchers explored various strategies that we review here.

Out-of-core algorithms. Out-of-core computation has been proposed to process large datasets for specific operations, mainly in linear algebra. Toledo [9] surveys such out-of-core for dense linear algebra operations. This appears also in sparse linear algebra: Demmel et al. [10] propose to reduce the amount of data transfers both between processing elements and from/to the main memory. Direct sparse solvers are known to produce large amount of temporary data, which makes out-of-core computing the only solution to factorize very large sparse matrices [3]. Marchal et al. [11] also focus on reducing data transfers for task trees arising in sparse direct solvers. Such algorithms are however tailored to specific application cases, while task-based runtime systems aim for generic-purpose designs.

Data locality in Big-Data frameworks. Data locality is recognized to be the key for performance in big-data analysis as well. Among others, the MapReduce framework tries to allocate tasks close to their data. It uses the Hadoop Distributed File System (HDFS), which splits files into large chunks of data and distributes them across nodes. By default, each block is replicated three times: two replicas are stored on the same cluster rack, while the third one is sent on a remote rack. The original motivation for replication is data resilience, since replication trivially limits the risk of losing a file both in the case of a node failure and in the case of a rack failure. However, it also allows the MapReduce scheduler to take advantage of the replication when allocating tasks: whenever a node has a slot available for a map task, a task using local data is allocated when it exists. A number of papers have studied the data locality in MapReduce, in particular during its "Map" phase. The work of Zaharia et al. [12], Xie and Lu [13] and Isard et al. [14] strive to increase the data locality by proposing better schedulers than the default one. Spark has a similar concern on data locality and allows users to fine-tune the balance between data locality and processing efficiency [15].

Scheduling and data locality in runtime systems. As outlined in the introduction, runtime systems like OmpSs [4], PaRSEC [5], or StarPU [6] are increasingly popular to cope with the complexity of modern computing platforms.

In this context, some runtimes have been striving to improve data locality for better performance. For example, runtime systems such as OmpSs and XKaapi [16] rely on work-stealing for load balancing. XKaapi also gives some guarantee on data locality [17] by providing a lower bound on the number of data accesses required by its scheduler. They also show that their locality-guided work-stealing policy performs significantly better than standard work-stealing.

We will use a similar work-stealing policy to evaluate our own strategies with the LWS scheduler introduced in Section 4.3.

On the contrary, the StarPU runtime system automatically calibrates performance models to predict task execution times. Based on these predictions, the DMDAS scheduler (presented below in Section 4.2) schedules tasks on the resource on which they are expected to complete at the earliest, which also takes data transfers into account. These predictions, however, only rely on the current state of the memory, and do not take into account its limited size: when some new data is loaded in memory, other data may be evicted, which is not taken into account and may lead to incorrect predictions.

Legion [18] allows the user to express locality thanks to data regions, and to provide an application-specific optimized data mapping. This mapping can be provided dynamically, to be able to adapt with machine changes and application irregularity. To some extent, the DARTS strategy could be reimplemented on top of this mapping interface. This interface would however not let DARTS retain tasks from execution (to privilege locality over priorities). An additional API interfacing would also be required to implement the eviction policy described in Section 5.4, required for best performance. PaRSEC, previously known as DAGuE [19], uses a different DAG representation, compared to StarPU and other runtimes: it uses parameterized task graphs [20]. The advantage of such a model is the concise representation of the DAG: each task is an algebraic representation that indicates which type of task is to be executed after a task is finished. Thus, the memory required for the DAG is only relative to the number of different task types. However, PaRSEC strategies do not specifically address the problem of scheduling under memory constraints. We compare our proposed scheduler to a PaRSEC scheduler presented in Section 4.4.

The Python-based Parla runtime [21] provides special data wrappers (Parla Arrays) which allows flexible memory management. Data locality is then considered when scheduling computations through a cost function that mixes the time required for moving data and the load of each node. This is very similar to the DMDAS scheduler mentioned for StarPU. Additionally, Parla does not specifically manage limited memory.

Data distribution can also be managed with modern high-level libraries. PHAST [22] or SYCL coupled with the Celerity API [23] can automate parallelization while leaving room to the programmer to direct data distribution on nodes. However, it must be manually programmed and does not specifically address memory limitation.

Hence, no existing runtime system is able to automatically deal with both

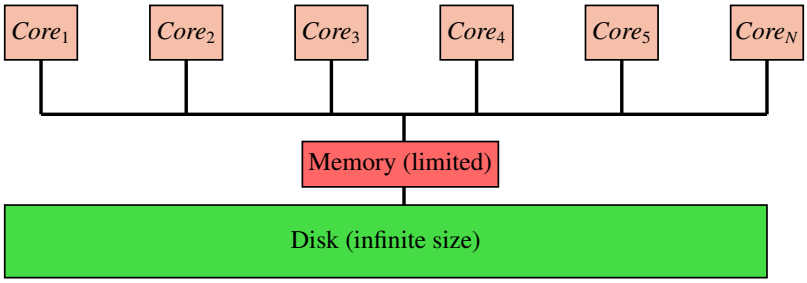


Figure 1: Platform topology of a multicore CPU with shared memory.

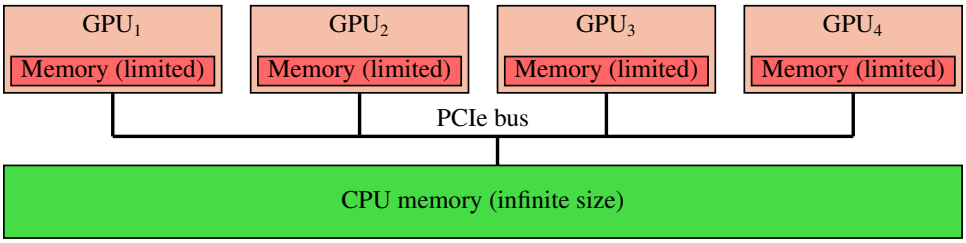


Figure 2: Platform topology of multiple GPUs with distributed memory.

data locality and limited memory for linear algebra computations.

3. Problem statement

We present here the model of the computing platform used when presenting the algorithms below, and we precisely state the objectives of a memory-aware scheduler for a runtime system.

3.1. Computing platform model

We concentrate in this paper on two different memory-limited architectures, namely a shared limited memory (Figure 1) and a distributed limited memory (Figure 2). We consider that the shared-memory setting is a special case of the distributed setting, as it includes a single memory, used by multiple cores which can be viewed as a single more efficient, parallel processing element. Hence, we concentrate the description of the problem and the algorithms on the more general distributed case, where the processing units are the GPUs and the large and slow storage is the main memory of the CPU. However, the problem and proposed algorithms can be easily translated for the shared memory case, as in the experiments reported in Section 7.3.

Each of the GPUs have their own memory with the same limited size M. They are all connected to the main memory of the machine through a shared communication link (PCIe bus) whose bounded bandwidth is shared by all the communications taking place simultaneously between the main memory and some GPUs. All input data originally reside on the main memory, whose size is assumed sufficient to store all the original and temporary data of the computation.

Note that in addition to the PCIe bus connecting all GPUs to the main memory, direct and faster connections may be available on some pairs of GPUs (such as NVidia NVLink). This may allow to load data in a GPU faster than from the main memory if the data is already on another GPU.

3.2. Task-based runtime scheduler

The application is described by the programmer as a task graph, where vertices represent tasks and edges represent data dependencies between tasks. The programmer provides the code for each task as well as the description of its input and output data, allowing the runtime system to assign tasks to processing units and to move data around when needed. Figure 3 provides a small example of such a task graph. At a given time of the computation, some of the tasks have been completed, and some tasks are not available for computation as their input data has not been computed yet (tasks with pale colors on Figure 3). The tasks available for computation, i.e., not yet processed but with all their predecessors completed, form a subset of independent tasks in the graph, called the *ready tasks* (bright tasks in the figure). Several of these available tasks depend on common predecessors (e.g., both GEMM_3_1_0 and GEMM_3_2_0 depend on the result of TRSM_3_0), which means that they share a common input data produced by the common predecessor. The runtime scheduler has the knowledge of this dependency pattern and can take advantage of this to improve data locality while mapping and scheduling tasks (e.g., by mapping GEMM_3_1_0 and GEMM_3_2_0 to the same GPU). We denote by $\mathcal{D}(T)$ the input data of task T. Data may have different sizes and tasks may have different processing times. To gather the processing times, StarPU performs automatic calibration [24, 25], usually done ahead of the computation. It measures the performance of tasks during executions and builds history-based performance models from previous runs. These performance models are then made available to StarPU's schedulers. Moreover, simulation results based on these models have been shown to be highly reliable [26].

The runtime system takes care of prefetching (loading a data a bit earlier so as to avoid waiting for unavailable data), marking tasks as available when their predecessors complete and managing task queues. The scheduler has the

```
Cholesky_decomposition(A):
for(k=0; k<N; k++)
   A[k][k]=POTRF(A[k][k])
   for(m=k+1; m<N; m++)
      A[m][k]=TRSM(A[k][k], A[m][k])
   for(n=k+1; n<N; n++)
      A[n][n]=SYRK(A[n][k], A[n][n])
      for(m=n+1; m<N; m++)
         A[m][n]+=GEMM(A[m][k],A[n][k])
```

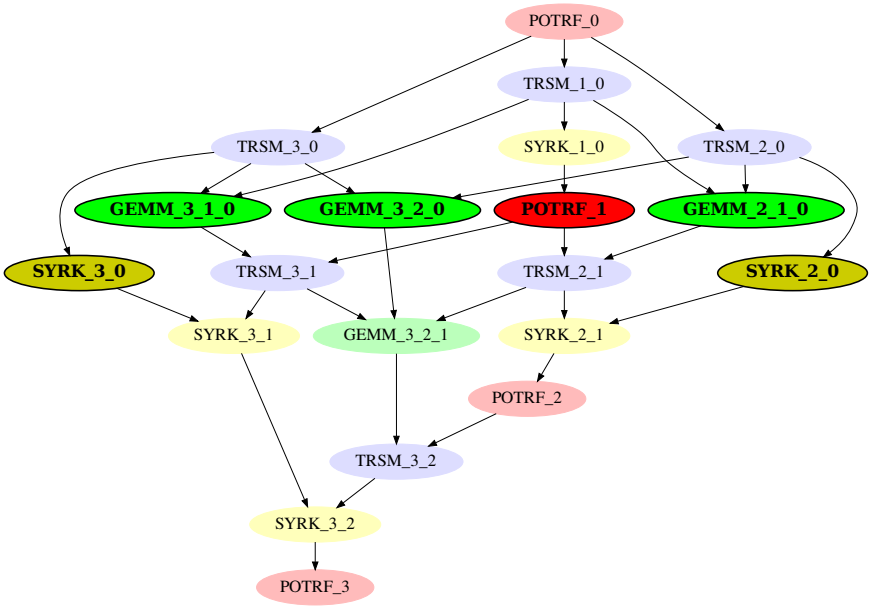


Figure 3: Pseudo-code for the Cholesky factorization (top) and corresponding task graph for $N = 3$ (bottom). Arrows represent data dependencies between tasks. Tasks with bright colors depict tasks available at a given moment in the computation; they are independent of each other.

responsibility to map and order tasks on each GPU. Given a set of ready tasks, when a GPU is idling, the scheduler must send a task to this GPU. If a GPU needs to load a data in order to compute a task and the memory is saturated, the scheduler has to evict a data from the GPU memory.

In this paper, we are focusing on optimizing the scheduler for data locality, hence we propose algorithms to **map tasks to processing units**, **order tasks on each processing unit** and **evict data** from the limited memory when needed.

In a previous study, we proved that the simpler problem of ordering tasks sharing input data to minimize data movement on a single processing unit was NP-complete [27], hence the more general problem studied in the present paper is also NP-complete.

Note that we concentrate here only on tasks' input data: in the case of linear algebra, there is usually at most as much output data as there are inputs. This includes in-place data updates. Therefore, their transfer does not take a longer time than transferring the data input. Additionally, optimizing for a reduced amount of input data transfers also leads to optimization for the data outputs. Data output is then not the driving constraint for efficient execution, as our experiments show. However, our model could be extended to account for large task outputs if necessary.

4. Existing runtime schedulers

In this Section, we present various algorithmic solutions that already exist in the literature to solve the partitioning and scheduling problem presented above. Some of these methods are greedy (Section 4.1), some first solve the partitioning problem, and then schedule the tasks on each processing unit (Section 4.2), some use work stealing to deal with load balancing and locality (Section 4.3), and some uses priority as a main focus point (Section 4.4).

4.1. The baseline: EAGER

The EAGER scheduler is a greedy scheduler that lets processing units (PUs) pick up tasks on demand from a shared queue. The queue is filled with tasks as they get released by dependencies. Hence, tasks are processed in the order of their release. Like all other presented schedulers, it prefetches data of tasks to be computed soon.

4.2. Dynamic scheduler of StarPU*: DMDAS*

DMDA or "Deque Model Data Aware" (Algorithm 1) is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing units in the StarPU runtime [28] (also called tmdp-pr). DMDA is based on the HEFT scheduler [29], which is already known to be an efficient scheduler. It computes for each processing unit PU_k the expected completion time $C(T_i, \mathrm{PU}_k)$ of the first task T_i in the queue of ready tasks, based on a prediction of the time required to transfer the data to the processing unit (*comm*) and of the task computation time (*comp*):

$$C(T_i, \mathrm{PU}_k) = \sum_{\substack{D_j \in \mathcal{D}(T_i) \\ D_j \notin mem(\mathrm{PU}_k)}} comm(D_j, \mathrm{PU}_k) + comp(T_i, \mathrm{PU}_k)$$

Note that the data transfer time is counted only if the data is not already in the memory of the processing unit. The task is then assigned to the processing unit

PU_k which minimizes $C(T_i, PU_k)$. Communication and computation times are predicted using a performance profile of the communication bus and processing unit which is calibrated beforehand. Tasks are assigned to processing units with this rule, one by one in the order of their release by the completion of their input dependencies.

DMDAS is a variant of DMDA that sorts tasks in the queue by priority order (as provided by the application). It is the default state-of-the-art scheduler used by the Chameleon library [30]. DMDAS also includes an additional *Ready* strategy (Algorithm 2): tasks are reordered at runtime in order to favor tasks with the most input data already loaded into memory.

Algorithm 1 Deque Model Data Aware heuristic (DMDA)

```
InMem ← ∅
while not all ready tasks have been assigned do
    T_i ← pop(readyTasks)
    Compute C(T_i, PU_k) using Eq. 4.2 for each PU_k
    Assign T_i to PU_k with smallest C(T_i, PU_k)
    for each D_j ∈ 𝒟(T_i) do
        Request data prefetch for D_j in PU_k
        Add D_j to memory(PU_k)
```

Algorithm 2 Ready reordering heuristic, called by PUs

```
Require: List L of assigned tasks
while L ≠ ∅ do
    Search first T ∈ L requiring the lowest amount of data transfers
    Wait for all data in 𝒟(T) to be in PU's memory
    Start processing T
```

In contrast to EAGER, DMDAS is a more advanced policy in terms of scheduling. It provides optimized execution thanks to its *Ready* strategy coupled with the DMDA algorithm and task priorities.

4.3. A work stealing policy: LWS

Work stealing policies exist on several runtimes systems like XKaapi [31], or libraries like TBB [32]. Work stealing has proven to be efficient in situations where partitioning and reducing communication is critical. We present here Locality Work Stealing (or LWS), a scheduler that combines work stealing for load

balancing, and locality to minimize communication. LWS is similar to the HWS scheduler introduced by Quintin et al. [33].

Each worker has a queue of tasks planned for future execution on the worker. LWS can deal with locality in two ways. First, when a task is released, it is queued on the worker that released it. Thus, a task and its descendants are all scheduled on the same worker. In most cases, the descendants of a task share at least one input data, so scheduling them on the same worker favors data reuse. Secondly, when a worker becomes idle, it steals a set of tasks from neighboring workers, starting from the end of their queue. There is a high chance that the inputs of these tasks have not been prefetched yet. This encourages workers to work on different input data. It thus favors distribution of different data to different workers, which increases the locality within each worker's task queue.

4.4. A priority-based scheduler from the PaRSEC runtime: AP

DPLASMA [34] is a well-known implementation of dense linear algebra operations for heterogeneous architectures. It uses PaRSEC [5], a dynamic runtime for architecture-aware scheduling of tasks. PaRSEC comes with different schedulers that all share two common aspects. First, they use the theoretical performance of CPUs and GPUs to balance the load assigned to each processing unit. Second, they use the knowledge of the DAG [35] to evaluate the cost of a task relative to its input, i.e. if multiple tasks use the same input data, the cost of a task will only be its computation time (without counting the data loading time) as its input data can be reused. Thus, multiple tasks that share input data have a higher probability of being computed on the same processing unit.

In our experiments, we use the Absolute Priority scheduler (AP) because of its affinities with priorities, an important feature for the applications we will be using. With AP, all processing units share a task-waiting queue that is sorted by priority, and each time a processing unit becomes available, it takes for execution a task from the head of the waiting queue. We also evaluated the default scheduler of PaRSEC: LFQ. We found that AP always gets slightly better performance on our applications. For this reason, we only show AP in the following experiments.

5. Improving the DARTS scheduler

Here we present our re-designed DARTS scheduler. We highlight the new features, but also include the main components of DARTS from [7].

5.1. Intuition

The principle of DARTS is to consider data locality before task allocation. The scheduling decision is based on the data already loaded into the memory of the processing units and the data used by the ready tasks. An ordered list of tasks is derived from it. The goal is to perform as many tasks as possible with the data available in the processing unit. Strategies presented in the previous section were favoring priority over locality: DMDAS and ParSEC AP sort tasks by priority, favoring progress on the task graph critical path over data reuse. On the contrary, DARTS can contradict priority when useful to favor data reuse. Trading priority for more locality is interesting when (i) a large number of tasks are available for computation at a given time, and (ii) computation of tasks from the critical path is not urgent because many tasks must be computed before the critical path becomes really critical. In such a case, there is enough parallelism available to occupy all the processing units, and favoring data locality can thus bring more benefits. DARTS aims to take advantage of this fact by considering as a primary factor the data that would bring the most data reuse if loaded, and as a secondary factor, in case of a tie between two data to be loaded, the one associated with the highest priority task.

5.2. Task flow

In order to grasp DARTS strategy, we need to understand the flow of tasks within the StarPU runtime as described in Figure 4. This figure uses the distributed memory case with GPUs because it is the more complete case, but the behavior is similar for shared memory with CPUs. The pink box represents the common StarPU core, and the blue elements are the actions of the scheduler. Tasks ready for execution are provided to DARTS through the *readyTasks* queue. When a GPU_k is idling, it will pull the head of the $taskBuffer_k$ queue of tasks. If that is empty, it will pull tasks from $plannedTasks_k$. And if that is empty, DARTS will be called to fill $plannedTasks_k$ using tasks from *readyTasks*. Note that tasks in $taskBuffer_k$ are in the StarPU common core and their data is being prefetched into the GPU memory. Thus, DARTS cannot access those tasks anymore. However, tasks in $plannedTasks_k$ can still be removed by DARTS as we will see in the eviction case (see section 5.4).

5.3. Strategy

The fundamental principle of DARTS is, when requested for tasks to put in $plannedTasks_k$ for PU_k, to first look for the best new data to load, that is, a data that will maximize the work that can be performed on PU_k.

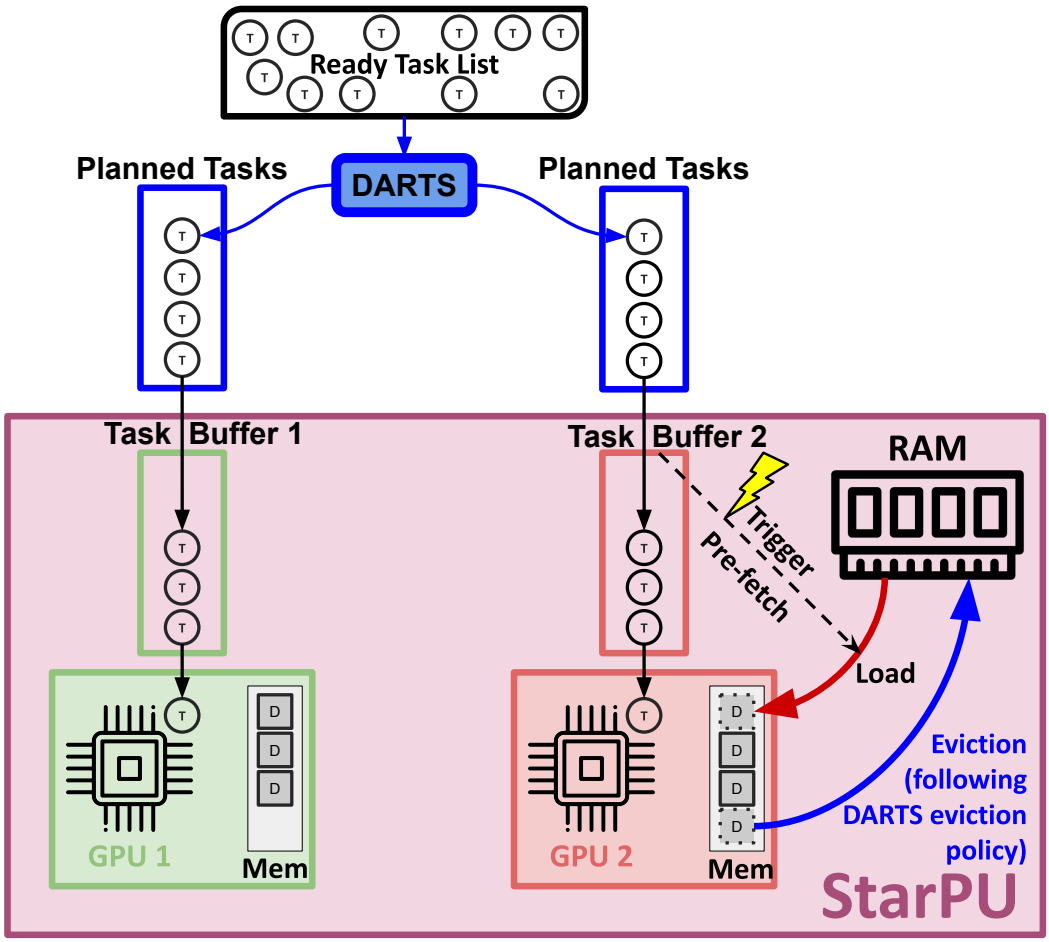


Figure 4: Flow of tasks in StarPU when using DARTS.

Our first improvement over the previous DARTS design comes from a complete re-design of the strategy of selection of the best data to load D_{opt}. We describe here the values required to select the optimal data D_{opt} (see Algorithm 3).

- $S_0(D)$: the set of tasks that depend only on D and some data already loaded in the memory of PU_k.
- $S_1(D)$: the set of tasks that depend only on D, some data already loaded in the memory of PU_k, and 1 additional data.
- *max_prio*(D): the highest priority of a task in $S_0(D)$ if $|S_0(D)| > 0$, otherwise the highest priority of a task in $S_1(D)$.
- *computation_time*(D): the sum of the durations of the tasks in $S_0(D)$.
- *task_left*(D): the sum of the durations of the tasks in *readyTasks* that use D as an input.
- *transfer_duration*(D): time required to load D to PU_k, taking into account the bus topology leading to PU_k.

Whenever some PU_k requests for some task to execute, we first look in the $dataNotInMem_k$ set (which initially contains all data) for the optimal data D_{opt} that, if moved into the memory of PU_k, would minimize the ratio between the time required to load D_{opt} on PU_k and the computation time of tasks that become "free", i.e., tasks that can be assigned and processed on PU_k without any additional data movement. Considering the transfer duration favors direct communication between GPUs (using NVLinks), which is much faster than using the PCIe bus. It is common for several data to have a similar ratio. This is especially the case when there is no data to enable "free" tasks. As a result, all data will have the same infinite ratio. To break ties, we first favor the data with the largest number of tasks in $S_0(D)$. Then, we break remaining ties by choosing the data that enables the computation of the highest-priority task T with one additional data load. In such a case, T depends on D_{opt}, on another data D not in memory, and possibly other data already in memory. Lastly, if we still have multiple candidate data for D_{opt}, we choose the data used by the task associated with the most remaining work by looking at both $S_1(D)$ and *task_left*(D). It is preferable to load data that will be used by many subsequent tasks, even if it does not bring locality immediately, as it has the potential to bring more opportunity for data reuse for subsequent tasks. Once D_{opt} is found, all the corresponding "free" tasks are assigned to $plannedTasks_k$.

Algorithm 3 DARTS scheduler on PU_k

When PU_k requests a new task

if $plannedTasks_k = \emptyset$ **then** ▷ We need to fill $plannedTasks_k$
  for each data $D \in dataNotInMem_k$ **do**
    Compute $S_0(D)$, $S_1(D)$, $max_prio(D)$,
$computation_time(D)$, $task_left(D)$ and $transfer_duration(D)$
    Compute $ratio = \frac{transfer_duration(D)}{computation_time(D)}$
  Choose the data D_{opt} with the smallest *ratio*, tiebreak in this order with $|S_0(D)|$, max_prio(D), $|S_1(D)|$ and $task_left$(D)
  if $|S_0(D_{opt})| > 0$ **then**
    Append $S_0(D_{opt})$ to $plannedTasks_k$
    Remove D_{opt} from $dataNotInMem_k$
  else if $|S_1(D_{opt})| > 0$ **then**
    Choose task T with highest priority from $S_1(D_{opt})$
    Append T to $plannedTasks_k$
    Remove the inputs of T from $dataNotInMem_k$
  else
    Choose task T with highest priority from *readyTasks*
    Append T to $plannedTasks_k$
    Remove the inputs of T from $dataNotInMem_k$
Return head of $plannedTasks_k$

The support of task graphs with dependencies requires supporting the dynamic addition of tasks in *readyTasks* when their dependencies are resolved. When such addition occurs, we dynamically update each $dataNotInMem_k$, to include data used by this new ready task but which was not used by any other ready task, and not loaded or planned for load on PU_k. The $dataNotInMem_k$ sets thus always contain exactly the set of data used by tasks that can be started (either in *readyTasks*, in some $plannedTasks_k$, or in some $taskBuffer_k$), which are not yet loaded on PU_k.

Besides, for some newly-released ready tasks, we can bypass *readyTasks* and directly push them to $plannedTasks_k$ when they are already "free". Since we know which tasks have been queued to each $taskBuffer_k$ and $plannedTasks_k$, we know the next data loading operations performed on PU_k. When a new task becomes ready, we can check if it will be free on PU_k, i.e., if it can be already be processed by some PU_k without any additional data load (because its inputs are already loaded or are waiting to be loaded). In such a case, we directly assign the new ready task to the corresponding $plannedTasks_k$. If several PUs qualify, we first consider the one with the fewest-queued tasks, to balance the load.

5.4. Eviction policy

In order to perform under memory constraints, DARTS needs a custom eviction policy, that matches its strategy. This strategy is named Least Used in the Future (LUF) and is detailed in Algorithm 4. The goal is to keep "free" tasks in *taskBuffer* and *plannedTasks* so as not to contradict the task order. The eviction policy considers all data currently in memory, and first tries to evict data that is not useful for any task in $taskBuffer_k$ (as those tasks are close to being executed on PU_k), and which is an input of few tasks in $plannedTasks_k$. If this is not possible, we apply Belady's rule [36]: evict the input data whose next usage in $taskBuffer_k$ is the furthest in the future.

The improved version of DARTS that we propose also brings a novelty to this eviction policy. We update $dataNotInMem_k$ for each PU_k after an eviction. This is essential for tasksets with dependencies, as the dataset must be dynamically managed to reduce the computational complexity of finding D_{opt}. If the evicted data is not used by any task in *readyTasks*, we remove it from all $dataNotInMem_k$. It will be added there again when a new ready task with this input becomes available. Otherwise, if any task of *readyTasks* uses the evicted data, we add the data to $dataNotInMem_k$ for each PU_k that does not hold it in memory (and is not scheduled to load it). This keeps $dataNotInMem_k$ up to date with the data that is currently required by ready tasks.

Algorithm 4 Eviction procedure LUF for DARTS on PU_k

for each data D in the memory of PU_k **do**
    $n_b(D) \leftarrow$ number of tasks using D in $taskBuffer_k$
    $n_p(D) \leftarrow$ number of tasks using D in $plannedTasks_k$
if the minimum value of $n_b(D)$ on any data D is 0 **then**
    Select V such that $n_b(V) = 0$ and $n_p(V)$ is minimum
else
    Select V the data whose next use in $taskBuffer_k$ is the furthest in the future
Remove tasks depending on V from $plannedTasks_k$
Evict V from memory, push it to $dataNotInMem_k$

5.5. Scheduling overhead study and optimizations

The main source of complexity in DARTS lies in finding D_{opt}. To reduce this complexity, we implemented two key optimizations.

Pointer based task and data mapping. Whenever a new task becomes available or is completed, we update bidirectional pointer lists: each task stores pointers to its input data, and each data maintains a list of pointers to the tasks that uses it. As a result, computing $S_0(D)$, $S_1(D)$ and $max_prio(D)$ can be done in a time proportional with the number of tasks using D: $\mathcal{T}(D)$, as we can directly access the tasks via the data's pointer list. And we only iterate over $dataNotInMem_k$ data. The complexity is thus $O(dataNotInMem_k \times \mathcal{T}(D))$. $dataNotInMem_k$ remains small in practice, especially in an online setting where data can quickly become "finished" as its corresponding tasks have been computed by other GPUs.

Refined mutex for parallel scheduling on multiple processing units. Mutual exclusion (or mutex) protects shared resources from being accessed by concurrent threads during overlapping time intervals. As mentioned above, the main complexity of DARTS comes from finding D_{opt}. Once D_{opt} is found, the tasks are moved from the *readyTasks* list to the *plannedTasks* list. This needs to be protected by a mutex. However, to find D_{opt}, the scheduler does not need to access a shared resource. Therefore, we introduce a refined mutex policy that lets concurrent processing units select D_{opt} during overlapping time intervals. The mutex is then used to protect task movement from *readyTasks* to *plannedTasks*. With our refined mutex policy, multiple processing units may select D_{opt} during the same time interval, greatly reducing the scheduling overhead.

As a result, the absolute duration of a scheduling decision in the worst case scenario (when most tasks are available simultaneously, typically at the beginning of the execution) has been observed to range between 100 and 200 µs. This overhead has no impact on performance in our GPU experiments, as the CPU handles the scheduling while not being used for computation (due to its much lower performance compared to GPUs).

6. Experimental settings

We detail below the experimental settings used to evaluate the schedulers discussed above. All schedulers were implemented in the StarPU runtime systems. The code of the schedulers and the applications is available online for reproducibility purpose [1].

6.1. Applications

All strategies mentioned above have been implemented in the StarPU runtime system [6], and evaluated both in distributed and shared memory. All schedulers use the LRU (Least Recently Used) eviction policy except for DARTS. The schedulers have been tested on two linear algebra applications: the Cholesky factorization ($A = L \times L^T$) and the LU factorization ($A = L \times U$) without pivoting.

Cholesky and LU are composed of tiled matrix multiplications (GEMM, requiring 3 input data), symmetric rank-k update for Cholesky only (SYRK, requiring 3 input data), triangular matrix equation (TRSM, requiring 2 input data), and Cholesky decomposition (POTRF, requiring 1 input data) or LU decomposition (GETRF, requiring 1 input data). We define task priorities using the bottom level, as introduced in [37]. The bottom level of a task corresponds to the minimum time required to complete the rest of the task graph starting from that task, assuming unbounded computational resources. This approach is used to assign priorities for DMDAS, LWS, and DARTS. The PaRSEC AP scheduler uses its own set of priorities (although we noticed no difference when using bottom-level priorities). Although we are running tests on Cholesky and LU factorizations, we made sure that our new DARTS performs as well as the previous version on the applications from our previous study.

We measure the obtained performance as the throughput of elementary computational operations performed per time unit (in GFlop/s), as well as the total volume of data transferred either between CPU and GPUs and between GPUs themselves in the case of distributed memory with GPUs, or between CPU and

[1] `https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/JPDC_reproducibility`

disk in the case of shared memory. When measuring the throughput, the cost of computing the schedule is considered. Each point in the following plots is the average of the performance obtained over 10 iterations. For most of the results, the standard deviation is less than 2%, thus we do not show error bars in the following graphs.

Although they are separate implementations for flexibility, our applications are identical to those of the Chameleon linear algebra library [38]. In the case of the PaRSEC AP scheduler, we use the DPLASMA implementation of the Cholesky decomposition. We made sure that both the StarPU implementation and the DPLASMA implementation of the Cholesky factorization use the same BLAS kernels with the same tile size.

6.2. Machines

We ran experiments using cuBLAS 11 single precision GPU kernels. Table 1 details the processing units used. We used three different generations of GPUs: Nvidia P100, V100, and A100. Since they were manufactured at very different times, we observe a gradual increase of their memory size, going from 16 GB per GPU to 40 GB. Similarly, NVLinks have various bandwidths, from single 10GB/s NVLinks for the P100 to single and dual 24 and 48GB/s NVLinks for the V100 and A100. The A100 has single 24GB/s NVLinks, but compensates with better PCIe bandwidth. The PCIe bus connects the GPU to the NUMA node, which contains the CPU and therefore access to main memory. While the V100 has the same PCIe bus bandwidth as the A100, it only has 2 buses for 4 GPUs, which creates possible contention. The A100 has as many buses as there are GPUs, so the full 12GB/s bandwidth is always available.

Figure 5 shows the topology of the node containing the V100 GPUs. We can see that the PCIe buses connecting the GPUs to the NUMA node have a bandwidth of 12GB/s for the connection to the first switch, and then 22GB/s for the link connecting directly to the NUMA node. This last link should theoretically have a bandwidth of 24 GB/s because it is connected to two 12 GB/s links, but the values we report in this figure and in Table 1 are the measured performance on the actual node and not the values reported by the vendor, which explains this anomaly. We can see that the bandwidth connecting the interconnect to the RAM is much higher, which makes the PCIe buses the main contention point. The PCIe buses are also the main bottleneck for the P100 and A100 nodes.

Table 1 also details the number of cores of each machine and their associated peak performance on the GEMM task. The shared memory settings of the CPU is detailed below in Section 7.3. We optimized the size of tiles used in the linear algebra computations to reach best peak performance for each architecture: we

use tiles of size 2880 on the V100 and A100 GPUs, and a smaller size of 1920 on the P100.

Table 1: Specifications of processing units used in the experimental evaluation.

	Nvidia P100	Nvidia V100	Nvidia A100	Intel Xeon E5-2620
Manufacturing year	2018	2019	2021	2012
Memory (GB)	16	32	40	32
NVLinks (GB/s)	10	24 or 48	24	/
PCIe Bus Bandwidth (GB/s)	13	12 (with contention)	12	/
Number of cores	3584	5120	6912	6×2 CPUs
Maximum GFlop/s on GEMM	6 700	14 000	18 000	15

7. Experimental evaluation

We present below the experimental evaluation conducted to compare the strategies presented above. We first detail results of the Cholesky factorization on a single V100 GPU. Then, we present results using three different types of GPUs: V100, A100, and P100, with and without memory constraint. Next, we discuss the results of LU decomposition on four GPUs, followed by the complete results with the three different GPUs. Finally, we discuss the results of the CPU experiments.

7.1. Cholesky factorization with GPUs

We present here a series of evaluations performed on GPUs using the Cholesky factorization.

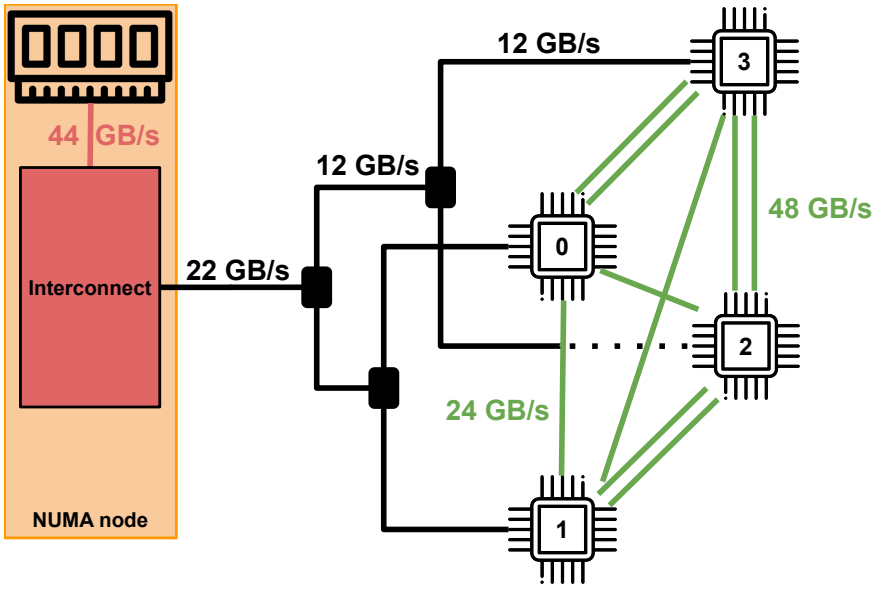
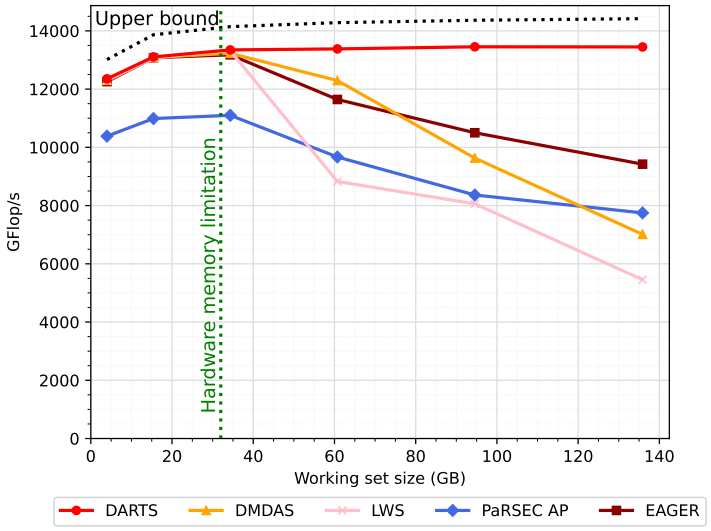


Figure 5: Topology of the node hosting the Nvidia V100 GPUs. The numbered processing units are V100 GPUs. The green lines are NVLinks, the black ones are PCIe busses. The black boxes are PEX8764 PCIe switches with 16GB/s PCI downlinks. The bandwidths shown were measured on the actual node used for our experiments.

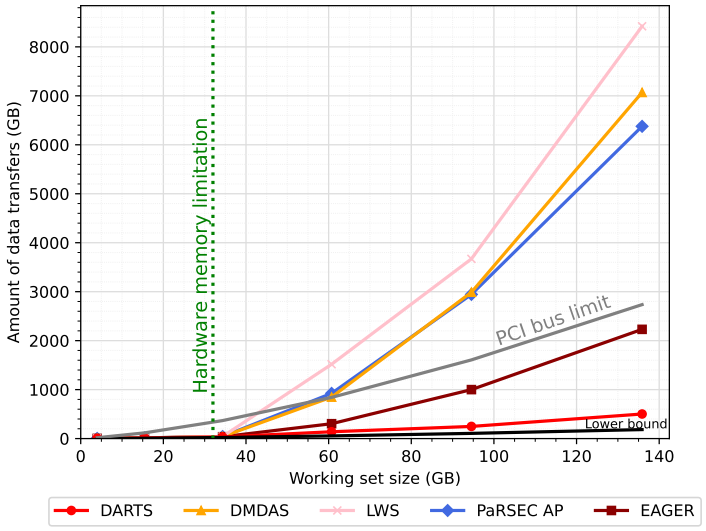
7.1.1. Overview

Figure 6 shows the results obtained by the various algorithms using one V100 GPU. Through calibration, StarPU creates history-based performance models for each kernel. StarPU also records which tasks are needed to compute an application. Using the kernel's performance model on the various available processing units and the application's task set, it computes through linear programming a theoretical lower bound on the execution time where dependencies are disregarded [40]. This bound is thus not always achievable but provides an interesting asymptotic goal: we try to make the execution as parallel as possible to get close to the bound. We plot on Figure 6a the theoretical upper bound on performance simply derived from the theoretical lower bound on the execution time.

The hard memory limit is marked with a green dotted line. We can also see on Figure 6b, with the solid gray line, the PCIe bus limit: a strategy exceeding this amount necessarily requires more time for the data transfers than the optimal time for computation. Figure 6b also depicts, with a solid black line, the lower bound on the communication volume required by the Cholesky factorization of an $N \times N$ symmetric matrix, as proven by Beaumont et al. [39].



(a) Performance. The black dotted line corresponds to the theoretical upper bound on performance computed by StarPU.



(b) Amount of data transfers from the CPU to the GPUs. The solid black line corresponds to the lower bound of communications as proven by [39]. The solid gray line is calculated from the amount of data to be transferred and the bandwidth of the PCIe bus, so it corresponds to the amount of data transfers that can theoretically be overlapped with computations.

Figure 6: Results on the Cholesky factorization with 1 Tesla V100 GPU. We vary the size of the working set on the x-axis by increasing the number of blocks of the tiled Cholesky factorization. For reference, the first point corresponds to a 15×15 matrix, while the last point corresponds to a 90×90 matrix. The green dotted line corresponds to the memory size of the V100 GPU.

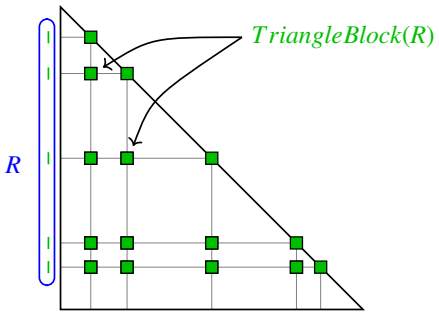


Figure 7: Representation of the tiles computed by a triangle block in an iteration of the communication optimal algorithm from [39]. R is a set of TRSM results that have already been computed. Triangle block(R) is the corresponding set of tasks to be processed to minimize IOs. Figure derived from figures in [39].

7.1.2. Optimal data access pattern

The authors of [39] prove (i) that it is not possible to have fewer IOs than the previous lower bound, and (ii) that there exists a sequential algorithm that matches such a result.

Figure 7 is an example of how the optimal schedule processes tasks on a given processing unit. The intuition is to partition the results of the TRSM tasks across multiple processing units and then compute as many SYRK and GEMM tasks as possible with such results. In Figure 7, we see on the left side, in blue, the results of TRSM operations that have already been computed on this processing unit. A node that has such results loaded into memory can compute the tiles in green (which are SYRKs or GEMMs) with a minimal amount of IOs. The communication-optimal algorithm would compute such *triangle blocks* associated with TRSM results already loaded on the GPU. Each GPU would compute tiles associated with different TRSM results, thus not replicating the data. Finally, such triangle blocks must be computed over multiple iterations. A scheduler that demonstrates a data access pattern in a triangular block that effectively reuses TRSM results across multiple iterations has the potential to significantly reduce IOs.

7.1.3. Single GPU case

From Figure 6a, we can see that all schedulers except DARTS greatly suffer from the limited memory of the GPU, as their performance plummets once the memory becomes a constraint (symbolized by the green line). With only one GPU, EAGER and LWS process tasks in their submission order. For EAGER, this results in poor progress on the critical path, which explains the lower per-

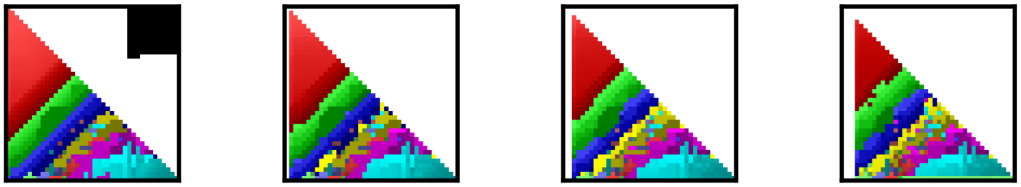


Figure 8: DMDAS's ordering on iterations 1 to 4 of the Cholesky factorization with 1 Tesla V100 GPU, with $N = 40$. Each task is depicted by a colored squared at its position in the matrix being factorized. The first 1000 processed tasks are in red, the next 1000 in green, then blue, yellow, magenta, cyan and orange. The shading within each color represents the processing order. The black area represents the amount of tiles that can be loaded in memory. The GPU's memory is limited to 2 GB for the visualization in order to mimic a very memory-constrained situation similar to the last point of Figure 6a (but with reduced matrix size).

formance. Another reason for EAGER's poor performance is that it does not reorder tasks for locality. As the working set grows, EAGER needs to load large amounts of data at once to compute the next row of the matrix. Consequently, each new task requires numerous data transfers, which stalls the computation before each task, while some other tasks do not require a data transfer, but will be only considered by EAGER after the burst of transfers. LWS has a large number of data loads, as shown in Figure 6b. LWS suffers from a pathological case of the LRU eviction policy. It loads as much data as possible into the GPU's memory and processes tasks of the Cholesky factorization one row of the matrix after the other. When the memory is full, it evicts the least recently used data. For a large matrix, the evicted data will still be used by subsequent tasks. This results in almost no data reuse, which leads to a lot of data transfers, creating a performance bottleneck.

DMDAS has more sustained performance on the 60 GB point but is far from the asymptotic goal on larger working set sizes. DMDAS does not consider the data set associated with each ready task. Thus, it does not reassign tasks according to new data loaded onto the GPU. Consequently, it cannot strike a balance between prefetch and eviction. The visualization in Figure 8 helps to understand these results. This corresponds to a similar situation from the last point of Figure 6a. Looking at the shading and color order, we can see that DMDAS processes tasks along anti-diagonals (because of task priorities). As an example, we can focus on the 1000 blue tasks. The first tasks to be processed are on the first upper left anti diagonal of iteration 1, then on iterations 2, 3, and 4. Then the tasks from the second blue diagonal of iteration 1 are computed, then those of iteration 2, 3 and 4. The tasks from the first anti diagonals have

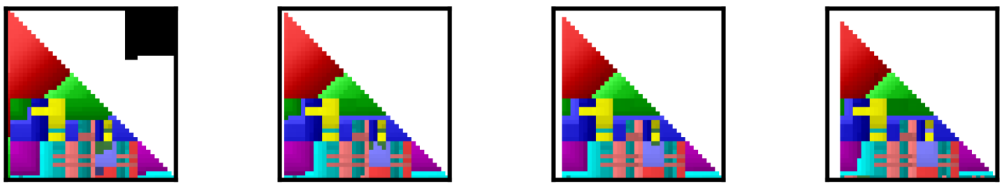


Figure 9: DARTS's ordering on iterations 1 to 4 of the Cholesky factorization with 1 Tesla V100 GPU, with $N = 40$. Same settings and visualization choices as Figure 8.

data affinity across the different iterations. However, the affinity on the rows and columns within an iteration can only be exploited if the memory can hold all the data needed for the first anti diagonals of each iteration. In a very memory-constrained case, after computing the first anti diagonal of each iteration, the GPU's memory is full. Thus, when the second blue anti diagonal is computed, all possible data that could have been reused has been evicted, resulting in multiple loads of the same rows or columns. This results in more data being transferred, as it can be seen in the Figure 6b. ParSEC AP schedules tasks similarly to DMDAS, as it uses the expected completion time of tasks and sorts them by priority.

DARTS maintains good performance with an increasing working set. DARTS also processes tasks in a diagonal while it can fit in memory (tasks in red on Figure 9) but then switches to what we call *DARTS triangle blocks*. We can notice such triangle blocks on the blue and magenta tasks (a triangle on the main diagonal preceded by a square of tasks of the same color on the first few columns). Note that DARTS triangle blocks are different from the optimal triangle blocks presented earlier because they do not necessarily group tasks from different zones, thus not forming a complete triangle. However, they have the benefit of reusing TRSM results to compute a maximum amount of tasks. Those triangles and the associated squares exactly fit in memory, share a lot of common data and are replicated over multiple iterations (up to $k = 8$) in order to progress on the critical path, while maximizing data reuse. DARTS forms these triangles because it can contradict priority with locality, as explained in Section 5.1. There is enough parallelism to feed all the workers without having to progress on the critical path. Thus, DARTS focuses on data reuse with its strategy instead of task priority, which creates these triangle blocks on the visualization. As explained earlier, accessing data with such a pattern allows to reduce communications with the RAM as shown in Figure 6b: DARTS has the lowest amount of data transfers and is under the bus limit. It means that theoretically, all transfers can be overlapped by a computation, which explains DARTS good performance.

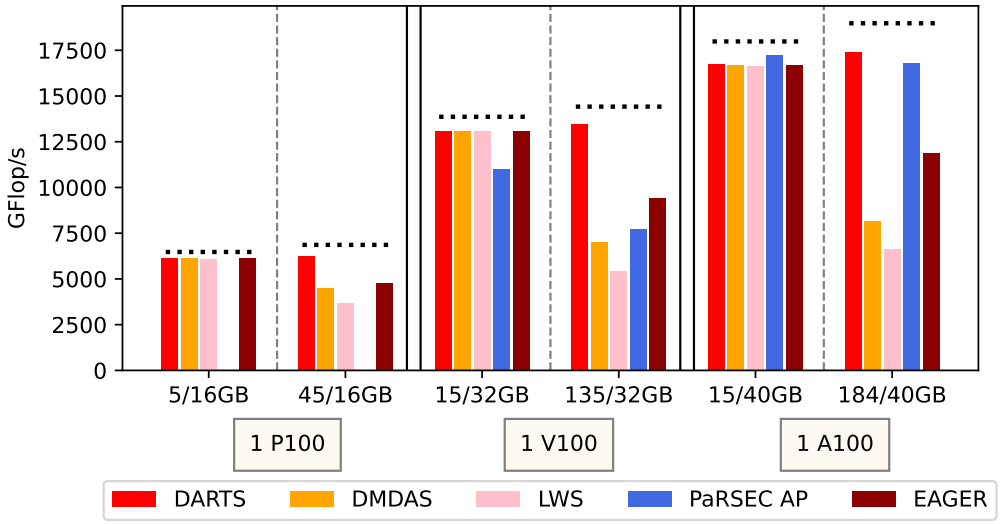


Figure 10: Performance with a single GPU on the Cholesky factorization. Each subfigure is a different experiment with a different GPU. The x-axis shows the working set size compared to the total memory of the GPU (e.g., 5/16GB denotes an experiment with a working set size of 5GB using a 16GB on-device memory). The solid vertical lines separate the experiments with different GPUs. The first half of each subfigure is a scenario where the working set size is smaller than the GPU memory. The second half is a memory-constrained situation. The black dotted line represents the theoretical upper bound on performance.

7.1.4. Results with different types of GPU

Figure 10 illustrates the performance of the different schedulers across different GPUs (P100, V100, and A100) and memory constraints. The experiments with P100 GPUs do not show ParSEC AP results: The P100 is a relatively slow GPU compared to its associated CPUs on the node, so ParSEC AP used CPUs in addition to GPUs for this experiment. Since we could not disable the CPUs, it was not a homogeneous configuration, so we could not compare the results.

Figure 10 illustrates that, for experiments where the full workload fits in memory, the majority of schedulers yield comparable outcomes. This is to be expected, since even the order in which tasks are submitted can lead to near-optimal performance as long as transfers are overlapped with computation, which is the case for all strategies here. For experiments with a working set size that exceeds the GPU memory, DARTS is proven to be the best strategy because it can reduce data transfers, similar to Figure 6b. However, on the A100 GPU, ParSEC AP and DARTS are almost equivalent, even after the memory constraint. This is due to two differences from the other compute nodes. First, the A100 GPU has more

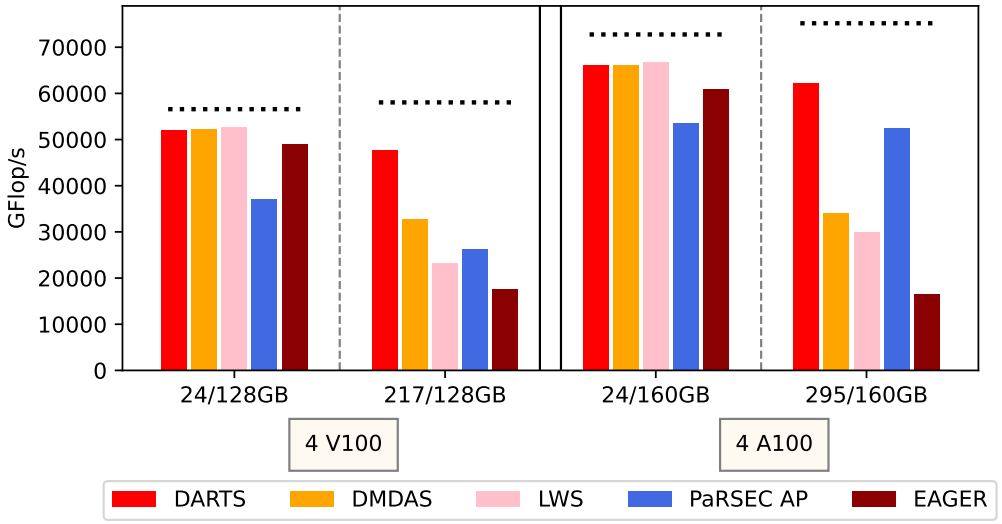


Figure 11: Performance with four GPUs on the Cholesky factorization. Same settings as Figure 10 excepts the memory size now considers the cumulated memory of all GPUs.

memory than the other GPUs. Second, the compute node containing the A100 has as many PCI buses as there are GPUs. As a result, there is no contention on the buses to transfer data between the CPU and the GPUs, unlike on the compute node containing the V100 GPUs. (see Table 1). With these two improvements compared to the V100s, the benefit of data locality is reduced because transfers can be done quickly and the GPU's memory can hold more data, leaving less opportunity for situations like the pathological case LWS suffered in Figure 6b. However, on the V100, with less memory and more contention, DARTS is better than ParSEC AP.

7.1.5. Results with multiple GPUs

Figure 11 illustrates the performance with four GPUs, while Figure 12 is the same experiment but shows the amount of data transfers. In Figure 12, we consider the transfers from CPUs to GPUs, as well as the transfers between GPUs via NVLinks. The memory size now refers to the cumulative memory of the four GPUs. The results cannot be displayed on the P100 GPUs because we did not have access to a machine with more than 2 P100 GPUs.

The performance gap between DARTS and the other strategies is more pronounced with four GPUs than with a single GPU configuration. This is particularly clear when compared to ParSEC AP and EAGER. With multiple GPUs, some data must be replicated to different GPU memories, which greatly increases

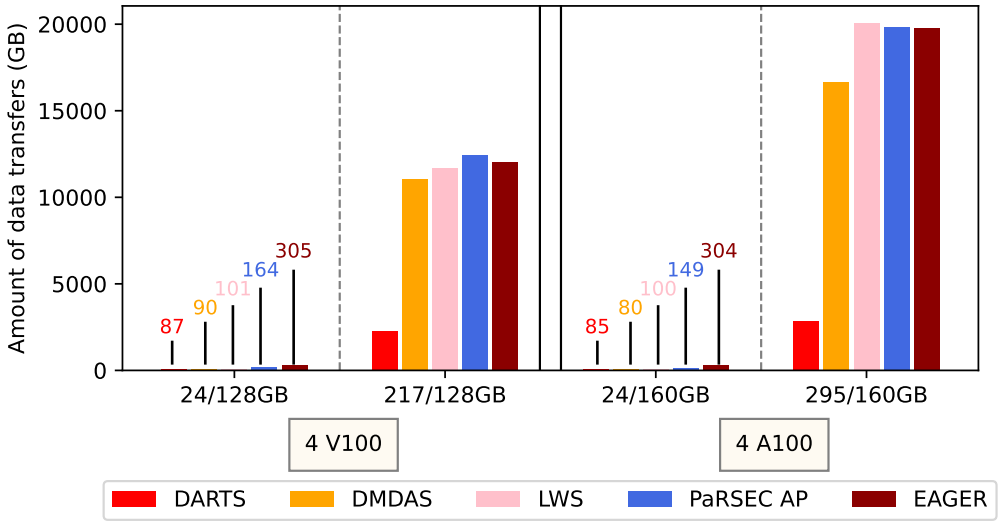


Figure 12: Amount of data transfers from CPU to the four GPUs and between GPUs on the Cholesky factorization for the experiments of Figure 11.

the total amount of data transfers, but it is necessary to increase parallelization. Normally this is not a problem, but with a memory constraint this replication greatly reduces performance because each GPU will suffer the same performance loss as seen with a single GPU. We can see that the amount of transfers on Figure 12 is very important for the other strategies compared to DARTS.

For small matrices that fit in the cumulative GPU memory, we see in Figure 11 that LWS achieves the best performance for both V100s and A100s by a small margin, but still notable for such small workloads. By stealing work from neighboring GPUs, LWS is able to greatly increase transfers using NVLinks, which are much faster than transfers using CPU memory. With four GPUs, the opportunity for such transfers is much more important, which explains this performance. This explains why in Figure 12, even with more transfers than DARTS, LWS achieves slightly better throughput on the experiments where the dataset fits in GPU memory. On both V100s and A100s, for larger matrices, LWS and EAGER have much more data transfers and thus much lower throughput. They do not take memory constraints into account and therefore do not favor data reuse. Similarly for ParSEC AP and DMDAS, the low throughput on the V100s and A100s experiment is associated with high transfer rates on Figure 12, as in the case of the single GPU experiment. Whenever memory is limited, DARTS consistently achieves the best performance.

Figure 13 represents the task order with two V100 (for clarity) using DARTS

(a) Task processed by GPU 1.

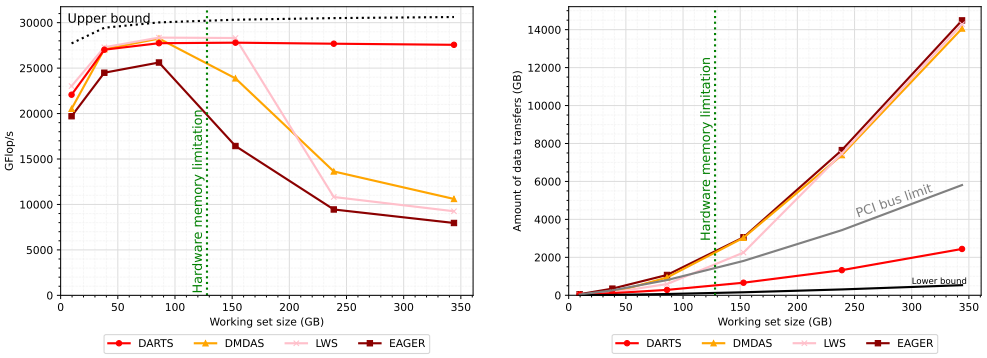
(b) Task processed by GPU 2.

Figure 13: Same settings and visualization choices as Figure 8.

on the first iteration of the outer-loop of the Cholesky algorithm. We can still find those *triangle blocks* that allows us to reduce data transfers, as can be seen on Figure 12. Moreover, there is a good distribution of the data load on the 2 GPUs. When choosing the data to load D_{opt}, DARTS uses a ratio between transfer time and computation time of task using D_{opt}. This favors the selection of a data associated with as many unprocessed tasks as possible. Thus two distinct processing units have a low probability to select the same data as the total work associated with a data is reduced after some of its task is scheduled on another processing unit. This encourages GPUs to work on different datasets, further reducing the total amount of data to load by minimizing the replication of data on multiple GPUs. The sustained performance of DARTS with both one and four GPUs shows that it is able to adapt to different numbers of processing units without having to change its strategy.

7.2. LU factorization with GPUs

We discuss here the results obtained on the LU factorization. Figures 14 depicts the performance and amount of data transfers on the LU factorization with four GPUs. It should be noted that we do not present results for PaRSEC AP as the DPLASMA implementation of LU does not make use of the cuSolver library to compute GETRF kernels of the LU factorization: ParSEC AP uses a much slower version of this kernel, and thus a fair comparison with other schedulers is not possible. The EAGER scheduler processes tasks in their submission order, while LWS sorts them by priorities. In both case, a tile loaded for a given computation cannot be reused on consecutive tasks when memory is limited. Consequently, these two schedulers end up with at least three times more data transfers than DARTS (see Figure 14b) once the memory becomes a constraint (after the green vertical line). DMDAS suffers from the same issue as with the Cholesky factorization: it is unable to balance prefetch and evictions, which results in a considerable amount of data transfers. DARTS exhibits more sustained



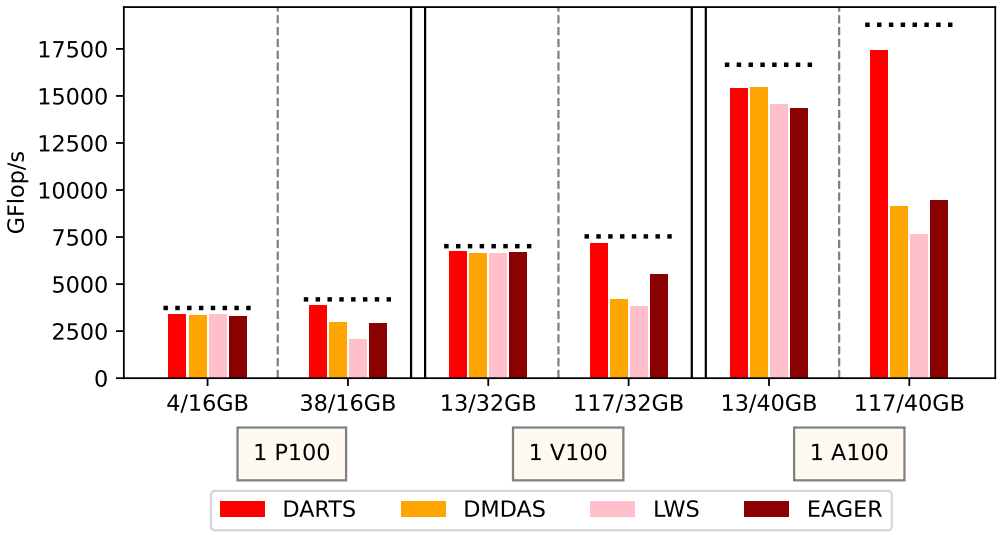
(a) Performance. The black dotted line corresponds to the theoretical upper bound on performance computed by StarPU.

(b) Amount of data transfers. The solid black line corresponds to the lower bound of communications as proven by [41]. The solid gray line is derived from the bandwidth of the PCIe bus and the working set size and represents the theoretical maximum amount of data transfers that can be overlapped with computations.

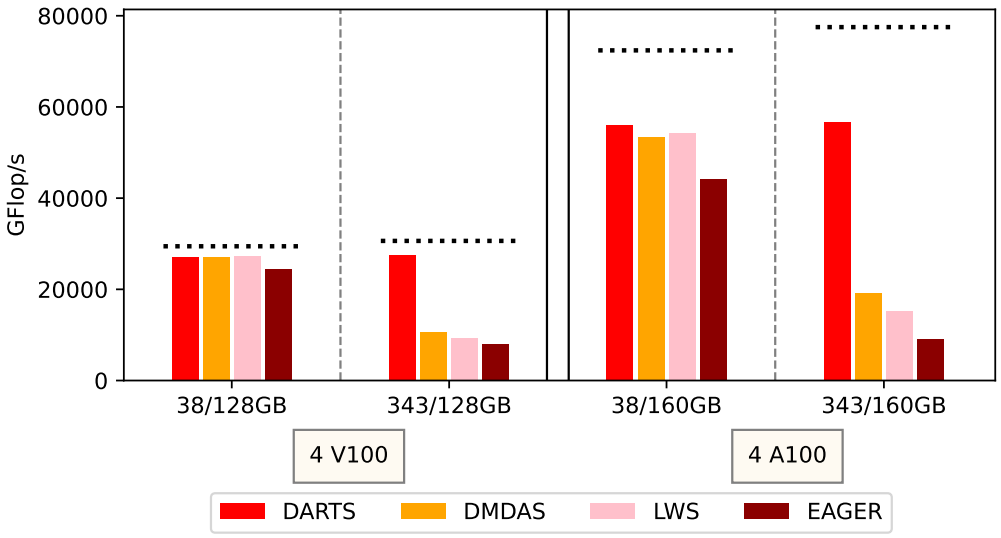
Figure 14: Results on the LU factorization with four Tesla V100 GPUs. We vary the size of the working set on the x-axis by increasing the number of blocks of the tiled LU factorization. For reference, the first point corresponds to a 12×12 matrix, while the last point corresponds to a 72×72 matrix. The green dotted line corresponds to the cumulated memory of the V100 GPUs.

performance: as is the Cholesky case, it is able to reduce data transfers by distributing the data on multiple GPUs and reusing data for consecutive tasks. This reaches 85% of the peak achievable performance, even with working sets twice as large as the cumulated GPU memory.

The LU and Cholesky factorizations are very closely related computations: LU can be considered a "double Cholesky". Because of the symmetry of the matrix in the case of Cholesky, LU requires twice as much data as Cholesky, but also includes twice as many computational tasks. Hence, both applications have the same communication-to-computation ratio. Similar data reuse patterns can be found for Cholesky and LU: building square blocks of GEMMs increases data reuse. For Cholesky, however, one must exploit the symmetry of the matrix to increase data reuse, which results in more complicated patterns. Such a solution is much harder to create with a dynamic scheduler. Given that DARTS is able to produce good data locality patterns on Cholesky, it is not surprising that it is able to reduce data transfers even more with LU. Figure 14b plots the minimum amount of IO required by LU factorization, as computed by Olivry et al. [41, Table 2], and we see that DARTS can get close to the minimum amount of IOs: DARTS has 4.8 times more data transfers than the lower bound and has 2.4 times less transfers than the PCIe bus limit. As a result, it is able to completely overlap

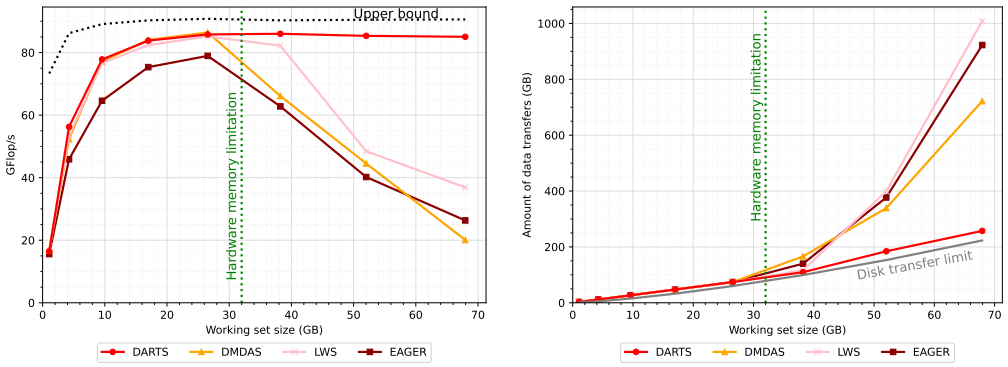


(a) Performance with a single GPU on the LU factorization.



(b) Performance with four GPUs on the LU factorization.

Figure 15: Performance achieved with one or several GPUs on the LU factorization. *X*/*Y*GB considers a scenario with a working set of size *X* GB for a (cumulative) GPU memory of *Y* GB. For each hardware setting, the left part presents a situation where the working set size is smaller than the GPU memory. The second half is a memory-constrained situation. The black dotted black represents the theoretical upper bound on performance.



(a) Performance.

(b) Amount of data transfers.

Figure 16: Results of the LU factorization with two Intel Xeon E5-2620 6-cores CPUs. Shared memory of 32 GB.

communications with computations, resulting in near-optimal performance with one or multiple GPUs. This explains why the performance difference with other schedulers is more significant in the LU case than in the Cholesky case.

Figures 15a and 15b illustrate the outcome of the experiment with the different GPUs. As previously observed, when the working set is smaller than the cumulated memory of GPUs, DARTS yields comparable results to the other schedulers. This is a satisfactory result, as it demonstrates that DARTS, despite its initial design to address memory-limited scenarios, is capable of achieving comparable performance to state-of-the-art solutions in more traditional cases. When the application dataset is significantly larger than the cumulated GPU memory, DARTS largely outperforms other schedulers. Again, compared to the results on Cholesky, we observe that the difference between DARTS and the other schedulers is more significant. We also observe that with four A100 GPUs, all schedulers perform further from the theoretical peak compared to the V100s. This is primarily because the theoretical PCIe bus limit is much lower on the V100 nodes due to their higher computational power. As a result, data transfers have a more pronounced negative impact on performance. Moreover, task dependencies further reduce attainable performance relative to the theoretical peak. This is particularly significant on the A100s, where the increased computational power further widens the gap between ideal and actual performance.

7.3. LU factorization on a multi-core CPU

Our scheduler is able to consider any memory-limited system, and StarPU manages disk-CPU transfers as well as CPU-GPU transfers, which enables us

to perform the experiments on CPUs. Figure 16 reports experiments on two Intel Xeon E5-2620 6 cores CPUs, with 32 GB of shared memory, and disks that sustain an approximately 94 MB/s bandwidth. Each core may need to transfer data from disk at the same time, placing a high demand on a limited bandwidth. On may argue that this particular CPU is slower and carries a smaller memory than most standard CPUs. However, its 32 GB and 12-core configuration provides 2.6 GB of memory per core, which is a reasonable allocation in comparison to contemporary CPUs that offer more memory but also a greater number of cores. Hence, it allows us to run experiments of reasonable durations on a hardware setting with realistic memory-per-CPU ratio. The DARTS strategy, which is specifically designed to reduce data transfers, is used in this architecture to demonstrate that near-optimal performance can be achieved with a relatively modest memory size in comparison to the data input. The application scenario is the LU factorization and we use tiles of size 2880.

In this shared-memory setting, DARTS behaves in a similar way as it does with one GPU: it uses a single *plannedTasks* queue as well as a single *taskBuffer*. On both experiments, when the whole dataset fits in memory, DARTS has comparable performance with DMDAS and LWS. For datasets larger than the available shared memory, DARTS is the only strategy able to obtain close-to-optimal performance, whereas all other schedulers suffer from a large performance drop. In terms of data transfers shown in Figure 16b, DARTS is able to stay close to the disk bandwidth limit, symbolized by the gray curve. Thus, DARTS has time to overlap most communications with computations, explaining its good results.

To confirm that DARTS's overhead remains negligible, Table 2 shows an experiment in which we varied the tile size while keeping the working set size fixed at 38 GB. Smaller tiles lead to a higher number of shorter tasks, thereby increasing the scheduling frequency and making overhead more significant. However, DARTS maintains consistent performance across tile sizes. This demonstrates that scheduling overhead does not impact performance when tasks become more numerous and shorter.

7.4. Summary of the experimental evaluation

The results demonstrated that when the cumulated memory of the processing units is larger than the size of the application's input data, DARTS exhibited similar performance to those of state-of-the-art runtime schedulers. Furthermore, when the memory is too scarce too load all input data simultaneously, existing schedulers experience large performance drops, while DARTS is able to sustain close-to-peak performance, by scheduling tasks sharing data on the same resource and reusing loaded data for successive tasks. This behaviour

Table 2: CPU performance of DARTS with a fixed 38 GB working set size while varying the tile on an Intel Xeon E5-2620 (6-core) CPU. The first row corresponds to the sixth point in Figure 16.

Number of tasks	Tile size	Matrix size (in tiles)	GFlop/s
4,900	2880	24×24	85.60
6,930	2560	27×27	86.45
10,416	2240	31×31	80.18
16,206	1920	36×36	85.98
27,434	1600	43×43	86.22
53,955	1280	54×54	83.68
127,020	960	72×72	79.30

has been acknowledged on both Cholesky and LU factorization, which involved complex computation dependencies, on various GPUs and CPU settings. The results demonstrate that DARTS is a scheduler that can be used with different linear algebra applications, memory configurations, and architectures. When the memory is the primary limiting factor, DARTS is capable of achieving near-peak performance.

8. Conclusion

8.1. Summary

In this paper, we focused on the problem of scheduling tasks in runtime systems to handle very large datasets that do not fit into the memory of the processing units. We improved a scheduler for the StarPU runtime system, named DARTS, to cope with task graphs. Our scheduler is mainly focused on data locality but also takes task priorities into account. We performed experiments with two classical linear algebra operations: Cholesky and LU factorizations. Thanks to its modularity, on a single node, DARTS reaches very good performance for linear algebra applications in a large variety of situations, from multicore CPU with shared memory to multiple GPUs with distributed memory. Among available competitors, DARTS in the only scheduler to reach good performance in memory constrained scenarios, and it also ranks among the best ones in all settings.

8.2. Future works

Future works include taking advantage of the modularity of DARTS to design hierarchical schedulers for large distributed platforms. At a small scale, an instance of DARTS would be used at each memory level: one would distribute

the tasks and data to the various nodes, and each node would be equipped with an instance of DARTS to distribute and order the task on each processing unit. This would work well with nodes equipped with GPUs, as they are generally not too numerous in a single node. However, within nodes mainly composed of CPUs, DARTS would not be able to assign the tasks to the many individual cores without getting hit by overhead issues. Thus, in such a case, we aim to couple DARTS with other schedulers, such as work-stealing algorithms (like LWS). So, the goal would be to use DARTS to partition the task set across the different shared memory NUMA nodes, since they are not too numerous within a node, and then an LWS-like policy would use its load balancing capability to carefully order the tasks onto the different individual cores. At a large scale, with possibly hundreds of nodes, the complexity of DARTS could become an issue because the response time would not be fast enough to serve all the workers. One possible solution would be to use a static workload partitioning [42], which would assign a subset of tasks to each node, and then an instance of DARTS on each node would manage the memory constraint of each processing unit.

While DARTS focuses primarily on data locality, we believe that schedulers should ideally be capable of dynamically adapting their optimization objective (like load balancing, critical path progress, or locality) based on the current progress through the task graph. A future goal is to develop a hybrid scheduler that leverages DARTS's data-aware strategy when enough parallelism is available, thereby minimizing data transfers. And, as the execution progresses and critical path becomes more crucial to unlocking further parallelism and achieving peak performance, the scheduler would shift toward a priority-based strategy.

Acknowledgments

This work was supported by the SOLHARIS project (ANR-19-CE46-0009) which is operated by the French National Research Agency (ANR).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

We thank Lionel Eyraud-Dubois for his help in designing Figure 7 similar to his work in [39] and for his insightful feedback.

References

[1] N. R. Council, et al., Getting up to speed: The future of supercomputing, National Academies Press, 2005.

[2] Q. Cheng, M. Glick, K. Bergman, Chapter 20 - Optical interconnection networks for high-performance systems, in: A. E. Willner (Ed.), Optical Fiber Telecommunications VII, Academic Press, 2020. doi:https://doi.org/10.1016/B978-0-12-816502-7.00020-8.
URL https://www.sciencedirect.com/science/article/pii/B9780128165027000208

[3] E. Agullo, A. Guermouche, J.-Y. L'Excellent, A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory, Parallel Computing 34 (6) (2008) 296–317. doi:https://doi.org/10.1016/j.parco.2008.03.007.
URL https://www.sciencedirect.com/science/article/pii/S0167819108000495

[4] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, J. Labarta, Productive programming of GPU clusters with OmpSs, in: International Parallel and Distributed Processing Symposium, 2012.

[5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J. Dongarra, PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability, Computing in Science and Engineering 15 (6) (2013).

[6] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23 (2011). doi:10.1002/cpe.1631.

[7] M. Gonthier, L. Marchal, S. Thibault, Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems, in: IPDPS 2022 - 36th IEEE International Parallel and Distributed Processing Symposium, IEEE, Lyon, France, 2022, pp. 1–11.
URL https://hal.inria.fr/hal-03552243

[8] P. J. Denning, The working set model for program behavior, Communications of the ACM 11 (5) (1968) 323–333.

[9] S. Toledo, A survey of out-of-core algorithms in numerical linear algebra, in: External Memory Algorithms and Visualization, American Mathematical Society Press, 1999, pp. 161–180.

[10] J. Demmel, M. Hoemmen, M. Mohiyuddin, K. Yelick, Avoiding communication in sparse matrix computations, in: IEEE International Symposium on Parallel and Distributed Processing, 2008. doi:10.1109/IPDPS.2008.4536305.

[11] L. Marchal, S. McCauley, B. Simon, F. Vivien, Minimizing I/Os in Out-of-Core Task Tree Scheduling, International Journal of Foundations of Computer Science 34 (01) (2023) 51–80. arXiv:https://doi.org/10.1142/S0129054122500186, doi:10.1142/S0129054122500186.
URL https://doi.org/10.1142/S0129054122500186

[12] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling, in: European Conference on Computer Systems (EuroSys), ACM, 2010, pp. 265–278.

[13] Q. Xie, Y. Lu, Degree-Guided Map-Reduce Task Assignment with Data Locality Constraint, in: International Symposium on Information Theory (ISIT), IEEE, 2012, pp. 985–989.

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: Fair

Scheduling for Distributed Computing Clusters, in: Symposium on Operating Systems Principles (SOSP), ACM, 2009, pp. 261–276.

[15] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: a unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65. doi:10.1145/2934664.
URL https://doi.org/10.1145/2934664

[16] J. V. Ferreira Lima, T. Gautier, V. Danjean, B. Raffin, N. Maillard, Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures, Parallel Computing 44 (2015) 37–52.

[17] U. A. Acar, G. E. Blelloch, R. D. Blumofe, The data locality of work stealing, Theory Comput. Syst. 35 (3) (2002) 321–347.

[18] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and independence with logical regions, in: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012. doi:10.1109/SC.2012.71.

[19] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, Dague: A generic distributed dag engine for high performance computing, Parallel Computing 38 (1), extensions for Next-Generation Parallel Programming Models (2012). doi:10.1016/j.parco.2011.10.003.

[20] M. Cosnard, M. Loi, Automatic task graph generation techniques, in: Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, Vol. 2, 1995. doi:10.1109/HICSS.1995.375471.

[21] H. Lee, W. Ruys, I. Henriksen, A. Peters, Y. Yan, S. Stephens, B. You, H. Fingler, M. Burtscher, M. Gligoric, et al., Parla: A python orchestration system for heterogeneous architectures, in: SC '22: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2022.
URL https://userweb.cs.txstate.edu/~burtscher/papers/sc22.pdf

[22] B. Peccerillo, S. Bartolini, Phast - a portable high-level modern c++ programming library for gpus and multi-cores, IEEE Transactions on Parallel and Distributed Systems 30 (1) (2019) 174–189.

[23] P. Thoman, P. Salzmann, B. Cosenza, T. Fahringer, Celerity: High-level c++ for accelerator clusters, in: R. Yahyapour (Ed.), Euro-Par 2019: Parallel Processing, Springer International Publishing, Cham, 2019, pp. 291–303.

[24] C. Augonnet, S. Thibault, R. Namyst, Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures, in: 3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009), Delft, Netherlands, 2009.
URL https://inria.hal.science/inria-00421333

[25] E. Agullo, B. Bramas, O. Coulaud, L. Stanisic, S. Thibault, Modeling Irregular Kernels of Task-based codes: Illustration with the Fast Multipole Method, Research Report RR-9036, INRIA Bordeaux (Feb. 2017).
URL https://inria.hal.science/hal-01474556

[26] H. Casanova, A. Giersch, A. Legrand, M. Quinson, F. Suter, Versatile, scalable, and accurate simulation of distributed applications and platforms, Journal of Parallel and Distributed Computing 74 (10) (Jun. 2014).

[27] M. Gonthier, L. Marchal, S. Thibault, Locality-Aware Scheduling of Independent Tasks

for Runtime Systems, in: COLOC - 5th workshop on data locality - 27th International European Conference on Parallel and Distributed Computing, Lisbon, Portugal, 2021, pp. 1–12.

[28] C. Augonnet, J. Clet-Ortega, S. Thibault, R. Namyst, Data-Aware Task Scheduling on Multi-Accelerator based Platforms, in: Int. Conf. on Parallel and Distributed Systems, 2010.

[29] H. Topcuoglu, S. Hariri, M.-Y. Wu, Task scheduling algorithms for heterogeneous processors, in: Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99), 1999. doi:10.1109/HCW.1999.765092.

[30] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S. P. Thibault, Achieving high performance on supercomputers with a sequential task-based programming model, IEEE Transactions on Parallel and Distributed Systems (2017). doi:10.1109/TPDS.2017.2766064.

[31] T. Gautier, J. V. Ferreira Lima, N. Maillard, B. Raffin, Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures, in: 6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG), Berlin, Germany, 2013.
URL https://inria.hal.science/hal-00780890

[32] A. Robison, M. Voss, A. Kukanov, Optimization via reflection on work stealing in tbb, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, 2008. doi:10.1109/IPDPS.2008.4536188.

[33] J.-N. Quintin, F. Wagner, Hierarchical work-stealing, in: Euro-Par 2010 - Parallel Processing, 2010, pp. 217–229.

[34] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaeif, P. Luszczek, A. YarKhan, J. Dongarra, Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA, in: International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011.

[35] S. Moustafa, M. Faverge, L. Plagne, P. Ramet, 3D cartesian transport sweep for massively parallel architectures with PARSEC, in: IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 581–590.

[36] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems Journal 5 (2) (1966).

[37] C. Alias, S. Thibault, L. Gonnord, A Compiler Algorithm to Guide Runtime Scheduling, Research Report RR-9315, INRIA Grenoble ; INRIA Bordeaux - Sud-Ouest (Dec. 2019).
URL https://hal.inria.fr/hal-02421327

[38] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs, in: GPU Computing Gems, Vol. 2, Morgan Kaufmann, 2010.
URL https://hal.inria.fr/inria-00547847

[39] O. Beaumont, L. Eyraud-Dubois, M. Vérité, J. Langou, I/O-Optimal Algorithms for Symmetric Linear Algebra Kernels, in: ACM Symposium on Parallelism in Algorithms and Architectures, 2022.
URL https://hal.inria.fr/hal-03580531

[40] Theoretical Lower Bound On Execution Time, https://files.inria.fr/starpu/doc/html/OfflinePerformanceTools.html#TheoreticalLowerBoundOnExecutionTime, accessed: 2024-06-27.

[41] A. Olivry, J. Langou, L.-N. Pouchet, P. Sadayappan, F. Rastello, Automated derivation of parametric data movement lower bounds for affine programs (2019). arXiv:1911.06664.

[42] L. L. Nesi, L. M. Schnorr, A. Legrand, Communication-aware load balancing of the lu factorization over heterogeneous clusters, in: 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS), 2020, pp. 54–63. doi:10.1109/ICPADS51040.2020.00017.